REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
28-02-2011	final technical report	May 2007 - November 2010

4. TITLE AND SUBTITLE
Continuous Improvement of Deployed Software Systems

5a. CONTRACT NUMBER
FA9550-07-1-0210

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Ben Liblit

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Wisconsin–Madison
1210 W Dayton St.
Madison, WI

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-OSR-VA-TR-2012-0181

12. DISTRIBUTION/AVAILABILITY STATEMENT
approved for public release

13. SUPPLEMENTARY NOTES

14. ABSTRACT

We completed development of an adaptive binary instrumentor. We devised a taxonomy of adaptive instrumentation strategies, and a collection of several search heuristics within that taxonomy. Averaged across many applications, our best heuristic finds the top-ranked predictor while exploring only 40% of the program: a significant improvement over the 100% exploration required by prior approaches. Reducing instrumentation overhead has been a key goal of this research. The new instrumentor imposes just 1% overhead on a variety of medium-sized benchmarks, as contrasted with 87% overhead for sampling-based instrumentation and 300% overhead for complete binary instrumentation. No prior related work has even come within an order of magnitude of this 1% overhead mark.

We introduced a novel, non-intrusive and highly scalable mechanism for debugging high-performance computing (HPC) applications that identifies and refines task equivalence sets, then captures the relative logical progress of each task as a partial order. Our study shows that using relative progr

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(Include area code)*

Continuous Improvement of Deployed Software Systems
Final Technical Report

Ben Liblit
Computer Sciences Department
University of Wisconsin–Madison

Period Covered: 14 March 2010 – 30 November 2010

1 Adaptive Binary Instrumentation

One problem with earlier CBI approaches is that instrumentation is injected once, at compile time, and can never be changed after that point without shipping a whole new application. AFOSR's sponsorship is supporting the development of a more dynamic, more adaptive alternative. Changing instrumentation post-deployment requires advances in binary instrumentation technology as well as novel algorithms to guide adaptation. The goal is to have extremely lightweight instrumentation that iteratively, systematically, and *automatically* pursues bugs back to their root causes in complex, deployed applications. If successful, we will be able to find more bugs, more quickly, with lower instrumentation load (overhead) for any individual user.

We have now completed an adaptive binary instrumentor. Our instrumentor, which builds upon the DynInst project of Miller et al., has been tested on a large number of small- to medium-sized programs from known bug-hunting benchmark suites, such as the Siemens suite and the UNL Software-artifact Infrastructure Repository (SIR). The implementation is feature-rich, supporting a variety of experimental studies.

Reducing instrumentation overhead has been a key goal of this research. We measure the new instrumentor's overhead at 1% on a variety of medium-sized benchmarks, as contrasted with 87% overhead for sampling-based instrumentation and 300% overhead for complete binary instrumentation. This is an exceptionally encouraging result. No prior CBI instrumentation work has even come within an order of magnitude of this 1% overhead mark.

Of course, lightweight instrumentation is only useful if it still lets one find bugs. We have devised a taxonomy of adaptive instrumentation strategies, and a collection of several search heuristics within that taxonomy. Searches may proceed strictly forward from `main`, or may proceed backward (and in general, bidirectionally) from known failure points. A random heuristic serves as our sanity test, and we find that all guided approaches perform significantly better. Our TTest heuristic in particular shows a increase in predicate scores over successive iterations, indicating that it finds high-scoring bug predictors quickly while exploring less of the program than other approaches. In a broader study, averaged across many applications, the TTest heuristic finds the top-ranked predictor while exploring only 40% of the program: a significant improvement over the 100% exploration required by prior approaches.

The work described above has been published as: Piramanayagam Arumuga Nainar and Ben Liblit. "Adaptive Bug Isolation." In Prem Devanbu and Sebastian Uchitel, editors, *32nd International Conference on Software Engineering (ICSE 2010)*, Cape Town, South Africa, May 2010. ACM SIGSOFT and IEEE.

2 Temporal Order Analysis

High Performance Computing (HPC) applications continue to grow in complexity, often combining large numbers of independent modules or libraries. Further, they are using unprecedented processor counts. These trends significantly complicate application design, development, and testing. In particular, interactive debugging techniques, as implemented in tools like gdb, DDT, or TotalView, no longer work well since they must gather data across all modules or libraries and provide little assistance in identifying tasks in which errors arise.

Novel techniques for bug detection and isolation, like CBI or DIDUCE, target this problem and provide users with semi-automatic mechanisms to aid in discovering root causes in sequential programs. However, extending these approaches to parallel codes is not straightforward and requires a deeper understanding of the cross-task relationships and synchronizations. In previous work, our collaborators developed STAT, the Stack Trace Analysis Tool, which provides a first step into this direction by aggregating stack trace information across all computation nodes. STAT uses this information to form equivalence classes of tasks, which identify a small subset of processes that can be debugged as representatives of the entire application.

While often sufficient, stack traces can be too coarse grain for grouping tasks and for understanding the relationship between their execution state. This coarseness may miss critical differences or dependencies. Thus, STAT does not always allow bug isolation and root cause analysis. Instead, we must identify additional data that captures the relative execution progress in each task and that supports accurate mapping of the debug state across all tasks.

In recent AFOSR-sponsored work, undertaken jointly with Lawrence Livermore National Laboratory, we introduce a novel, non-intrusive and highly scalable mechanism that refines task equivalence sets and captures the progress of each task. This technique creates a partial order across the task equivalence classes that corresponds to their relative logical execution progress. For sequential code regions, our approach analyzes the control structure of the targeted region and associates it with an observed program location. For loops or other complex control structures, we use static data flow analysis, implemented in the ROSE source-to-source translation infrastructure, to determine which application variables capture relative progress. We then extract their run-time values in all tasks to refine the task equivalence sets and to determine their relative execution progress. Our static techniques significantly reduce the amount of run-time data needed for analysis.

Our methodology requires no source code changes; it analyzes the existing code and then uses the results to locate the relevant dynamic application state through standard debugger interfaces. Thus, our approach meets a critical debugging requirement: we can apply it to production runs, e.g., after the application aborts or hangs due to deadlock or livelock, thus manifesting a program bug.

Our study shows that relative progress significantly reduces the effort to locate tasks that manifest errors for a wide array of faults. In particular, we identify randomly injected errors in the Block Tridiagonal (BT) benchmark of the NAS Parallel Benchmarks and a real deadlock that AMG2006 exhibited at 4,096 tasks. Our evaluation demonstrates that our technique extends STAT to provide these benefits with moderate additional overhead.

The work described above has been published as: Dong H. Ahn, Bronis R. de Supinski, Ignacio Laguna, Gregory L. Lee, Ben Liblit, Barton P. Miller, and Martin Schulz. “Scalable Temporal Order Analysis for Large Scale Debugging.” In William Gropp, editor, *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC 2009)*, Portland, Oregon, November 2009. ACM SIGARCH and IEEE.

3 Sampling for Concurrency Bugs

Fixing concurrency bugs (or *crugs*) is critical in modern software systems. Static analyses to find crugs such as data races and atomicity violations scale poorly, while dynamic approaches incur high run-time overheads. Crugs manifest only under specific execution interleavings that may not arise during in-house testing, thereby demanding a lightweight program monitoring technique that can be used post-deployment.

Sponsored by AFOSR, we are developing Cooperative Crug Isolation (CCI), a low-overhead dynamic strategy for diagnosing production-run failures in concurrent programs. Following the Cooperative Bug Isolation philosophy, CCI monitors interleaving-related predicates at run time; leverages sampling to keep the run-time overhead low; and uses statistical models to process run-time information aggregated through many runs and many users and identify the root causes of production-run failures.

Applying the Cooperative Bug Isolation idea to crugs raises two major questions:

What types of predicates are suitable for crug diagnosis? Instrumentation must balance failure-predictive power and computational simplicity. Poorly-designed predicates may be unable to explain any crug failures. Yet costly predicates must use low sampling rates in order to provide performance guarantees, thereby delaying diagnosis. If the evaluation of a predicate relies on long and continuous monitoring, it may be unsuitable for sampling.

How can we sample predicates that are related to concurrency bugs? Previous CBI work made predicate sampling decisions independently in each thread at each execution point. CCI sampling is much more complex. It may require cross-thread coordination, because crugs involve multiple threads. It must also keep each sampling period active for some time, because crugs always involve multiple memory accesses. Proper sampling design affects both the number of predicates that can be collected as well as the correctness of the collected data.

There may be no single solution to all of the above challenges. We consider three different types of predicates together with new sampling strategies that support each. These three schemes offer different types of information that may help diagnose crug failures. They provide different trade-offs between performance and failure-predicting capability, and demonstrate different ways of sampling in concurrent programs.

Experimental results show that CCI is very effective, dramatically outstripping CBI, for crug diagnosis. Traditional CBI tools fail completely, providing no predictors for any of our buggy concurrent test subjects. The predictors reported by CCI, however, accurately point to the root causes of a wide variety of failures. Furthermore, CCI achieves this excellent failure diagnosis with small run-time overhead (mostly within 10%), thanks to its sampling mechanisms.

The work described above has been published as:

- Aditya Thakur, Rathijit Sen, Ben Liblit, and Shan Lu. “Cooperative Crug Isolation.” In Ben Liblit and Andy Podgurski, editors, *Proceedings of the Seventh International Workshop on Dynamic Analysis (WODA 2009)*, Chicago, Illinois, July 2009. ACM SIGSOFT.
- Guoliang Jin, Aditya Thakur, Ben Liblit, and Shan Lu. “Instrumentation and Sampling Strategies for Cooperative Concurrency Bug Isolation.” In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010)*, October 2010.